ROSE::FTTransform – A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research

Jacob Lidman*†, Daniel J. Quinlan†,
Chunhua (Leo) Liao†, Sally A. McKee*

Lawrence Livermore National Laboratory

CHALMERS

*Chalmers University of Technology, Sweden
†Lawrence Livermore National Laboratory, USA

LLNL-PRES-562817

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Outline

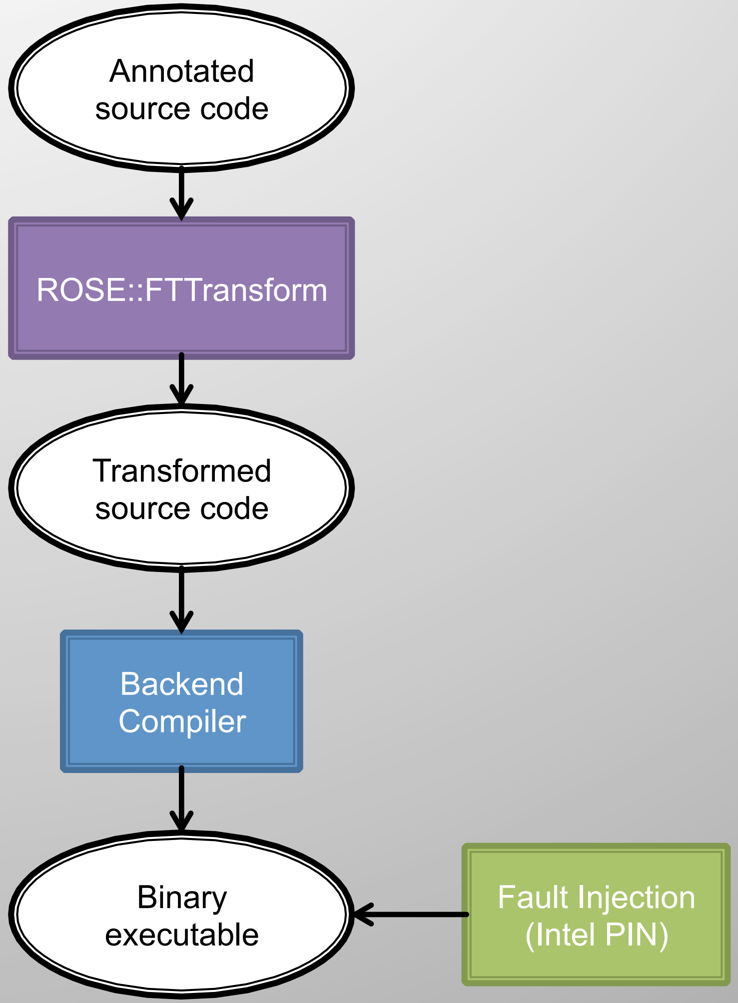
- Motivation
- Approach
- Results
- Summary
- Future work

Motivation

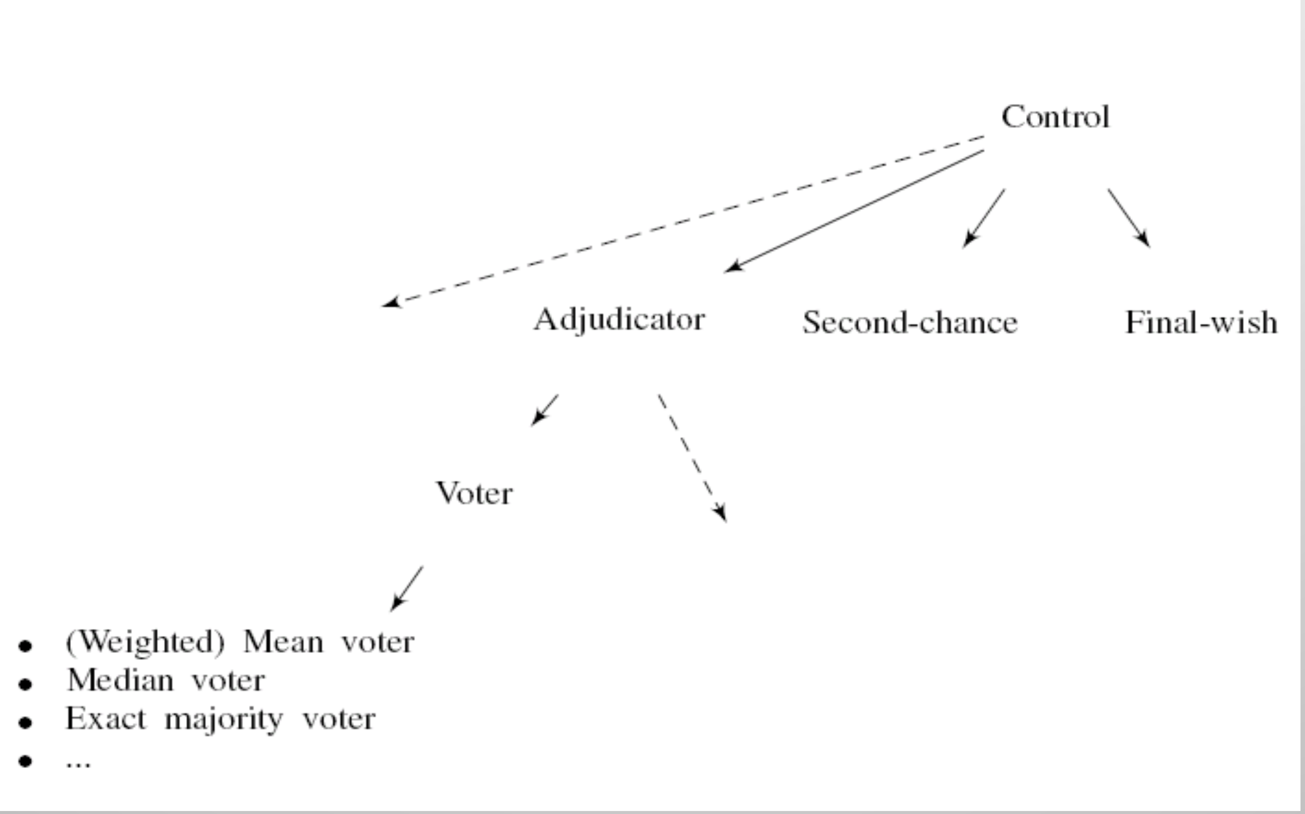
- Resilience: a big challenge for Exascale systems
  - Millions of processors/cores
  - Low-power processors/cores: power requirements
    - Increased sensitivity to internal/external events
    - Transient faults: going wrong without being noticed
  - Streamlined and simple processors
    - Cannot afford pure hardware-based resilience
- An attractive solution
  - Software-implemented hardware fault tolerance (SIHFT)
  - In house source-to-source compiler infrastructure: ROSE@LLNL

Approach: compiler-based transformations to add resilience

- Source code annotation (pragmas)
  - What to protect
  - What to do when things go wrong
  - Can be auto inserted later on
- Source-to-source translator
  - Fault detection: N-Modular Redundancy (NMR)
  - Connect to fault-handling policies
- Backend compiler: vendor compilers or GCC
  - binary executable
- Intel PIN: fault injection



Hierarchical structure of fault-handling policies



- Controller policies: e.g. Final-wish, Second-chance. Must be used with next-level policies
- Terminal policies: final decision about how to unify results. e.g adjudicators implementing voting strategies, mean, median, majority voting, etc.

Source code pragmas and semantics

	Final-wish	Second-chance
Pragmas syntax	#pragma resilience FT-FW (NEXT_POLICY) y =f (x);	#pragma resilience FT-SC(NEXT_POLICY, NUM_ITER) y =f (x);

Semantics	// N-Modular Redundancy
	y[0] = f(x);
	...
	y[N-1] = f(x) ;
	
	// Tentatively pick one as the final result
	y = PICK_RANDOM(y[0] , . . . , y[N-1]);
	// Fault detection
	if (!EQUALS(y[0] , . . . , y[N-1] ,y))
	{
	// Fault handling
	NEXT_POLICY;
	}
		for (int rl = 0 ; ; rl ++)
		{
		// N-Modular Redundancy
		y[0] = f(x) ;
		...
		y[N-1] = f(x) ;
		
		// Tentatively pick one as the final result
		y = PICK_RANDOM(y[0] , . . . , y[N-1]);
		
		// No Fault is detected?
		if (EQUALS(y[0] , . . . , y[N-1] ,y))
		break;
		// Reaching the limit of having a second chance ?
		else if (rl == NUM_ITER)
		// Fault handling
		NEXT_POLICY;
		}

Concerns for implementing source level N-Modular Redundancy

- ROSE::FTTransform's central idea:
  - Detects/handles transient processor faults via redundant execution of critical source code statements
    - — Naive implementation: duplication of N copies of computation
  - Feasible?
    - — back-end compilers have Common Subexpression Elimination (CSE)
  - Overhead?
    - — Nx times slower in worst case

Transformations: optimizer-proof code redundancy

```
/* Original Jacobi 1-D , 3-points computation kernel */
void kernel1()
{
  int i;
  for (i=1; i<SIZE-1; i=i+1)
  {
    d[i] =  0.25*c[i-1] + 0.5*c[i] +0.25*c[i+1];
  }
}
/* Transformed kernel with redundant computation */
void kernel2(double *c2 )
{
  double B_intra[3];
  int i;
  for (i=1; i<SIZE-1; i=i+1)
  {
    /* Baseline double modular redundancy (DMR) */
    B_intra[0]= 0.25*c[i-1]+0.5*c[i]+ 0.25*c[i+1];
    B_intra[1]= 0.25*c2[i-1]+0.5*c2[i]+ 0.25*c2[i+1];
    d[i]= B_intra[0];
    if (!equal(B_intra[0],B_intra[1],d[i] )
    {
      /* Additional N-2 redundancy and
         fault handling mechanism omitted here... */
    }
  }
}
...
/* call site doing pointer declaration and assignment */
 double *c2 = c;
 kernel2(c2);
```

Statement to be protected

Using an extra pointer to help preserve source code redundancy

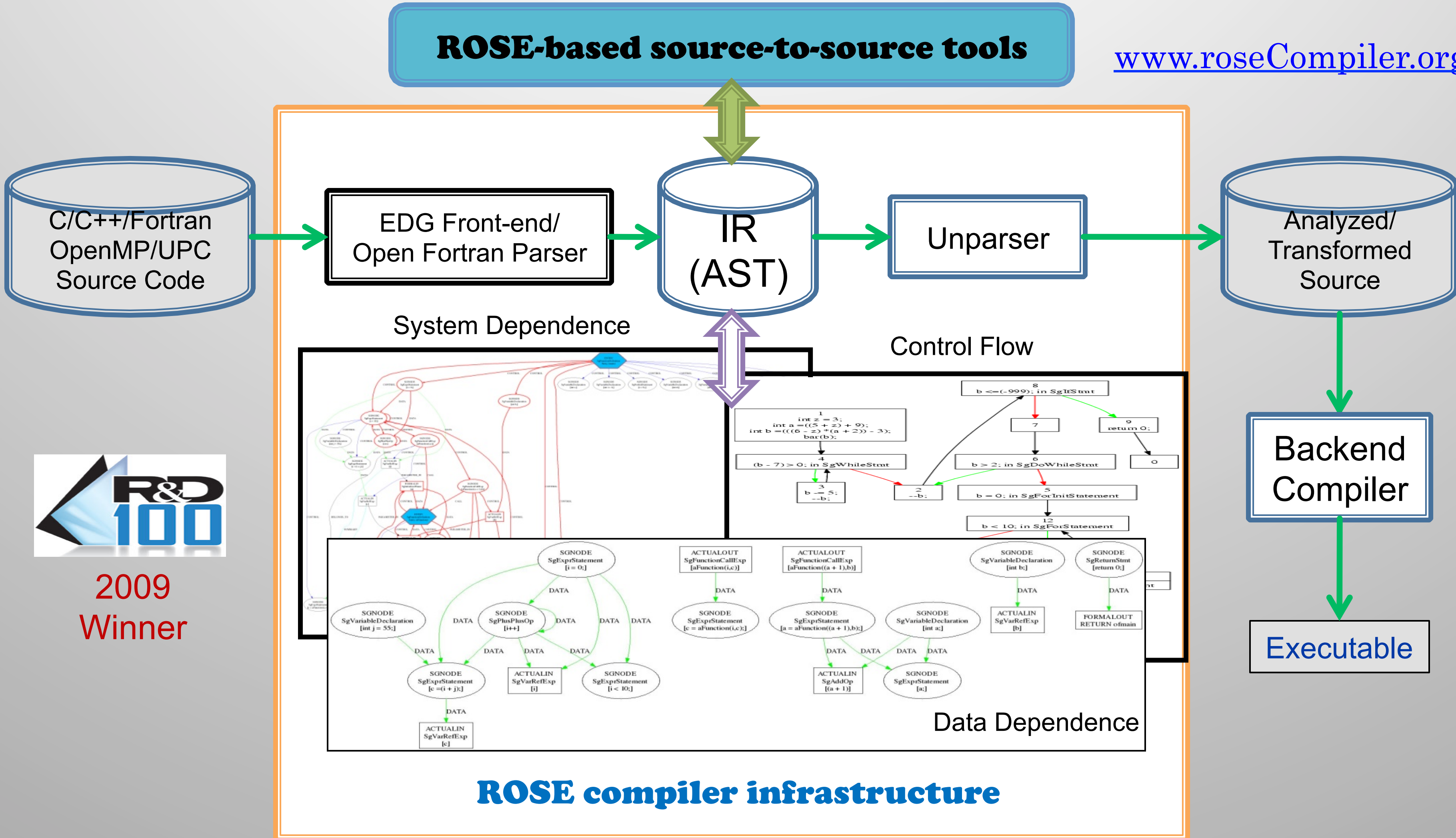
Transformations: reducing overhead for NMR

```c
/* Original Jacobi 1-D , 3-points computation kernel */
void kernel1()
{
  int i;
  for (i=1; i<SIZE-1; i=i+1)
  {
    d[i] =  0.25*c[i-1] + 0.5*c[i] +0.25*c[i+1];
  }
}
/* Transformed kernel with redundant computation */
void kernel2(double *c2 )
{
  double B_intra[3];
  int i;
  for (i=1; i<SIZE-1; i=i+1)
  {
    /* Baseline double modular redundancy (DMR) */
    B_intra[0]= 0.25*c[i-1]+0.5*c[i]+ 0.25*c[i+1];
    B_intra[1]= 0.25*c2[i-1]+0.5*c2[i]+ 0.25*c2[i+1];
    d[i]= B_intra[0];
    if (!equal(B_intra[0],B_intra[1],d[i] )
    {
      /* Additional N-2 redundancy and
         fault handling mechanism omitted here... */
    }
  }
}
...
/* call site doing pointer declaration and assignment */
 double *c2 = c;
 kernel2(c2);
```

Statement to be protected

Relying on baseline double modular redundancy (DMR) to help reduce overhead

Implementation of ROSE::FTTransform



2009
Winner

Results: necessity and effectiveness of optimizer-proof code redundancy

- Check if redundant computation can survive compiler optimizations
  - Jacobi 1-D 3-point kernel 1) original version, and 2) protected version using double module redundancy.
  - PAPI (PAPI_FP_INS): the number of floating point instructions for both versions
  - GCC 4.3.4, O1 to O3

Transformation Method	PAPI_FP_INS DMR/Orig. (O1)	PAPI_FP_INS DMR/Orig. (O2)	PAPI_FP_INS DMR/Orig. (O3)
Our method of using pointers	Doubled	Doubled	Doubled
Naïve Duplication	The same	The same	The same
Naïve Duplication buried within a basic block	The same	The same	The same

Results: performance overhead

- Performance overhead: DMR
  - Also good approximation for general NMR, excluding overhead from the incidental N-2 redundancy and fault-handling mechanism
  - Experimental environment
    - — 4-core AMD Opteron: L1 data: 64K, L2: 512K, L3 6M, 129GB Memory
    - — 64-bit SUSE Enterprise 11.1, GCC 4.3.4 (-03)
  - Benchmarks:
    - — three versions of Jacobi : 1D 1-point, 1D 3-point, and 2D 5-point
    - — Livermore loops*
  - Explore impact of latencies of the original codes (Jacobi):
    - — Data set size: arrays fitting into cache or not
    - — Iteration strides: 1 vs. 8
    - — Element sizes: single vs. double precision

* http://en.wikipedia.org/wiki/Livermore_loops

Results (cont.) — overhead

- Jacobi kernel:
  - Overhead: 0% to 30%
  - Minimum overhead
    - Stride=8
    - Array size 16Kx16K
    - Double precision
  - The more original latency, the less overhead of added redundancy

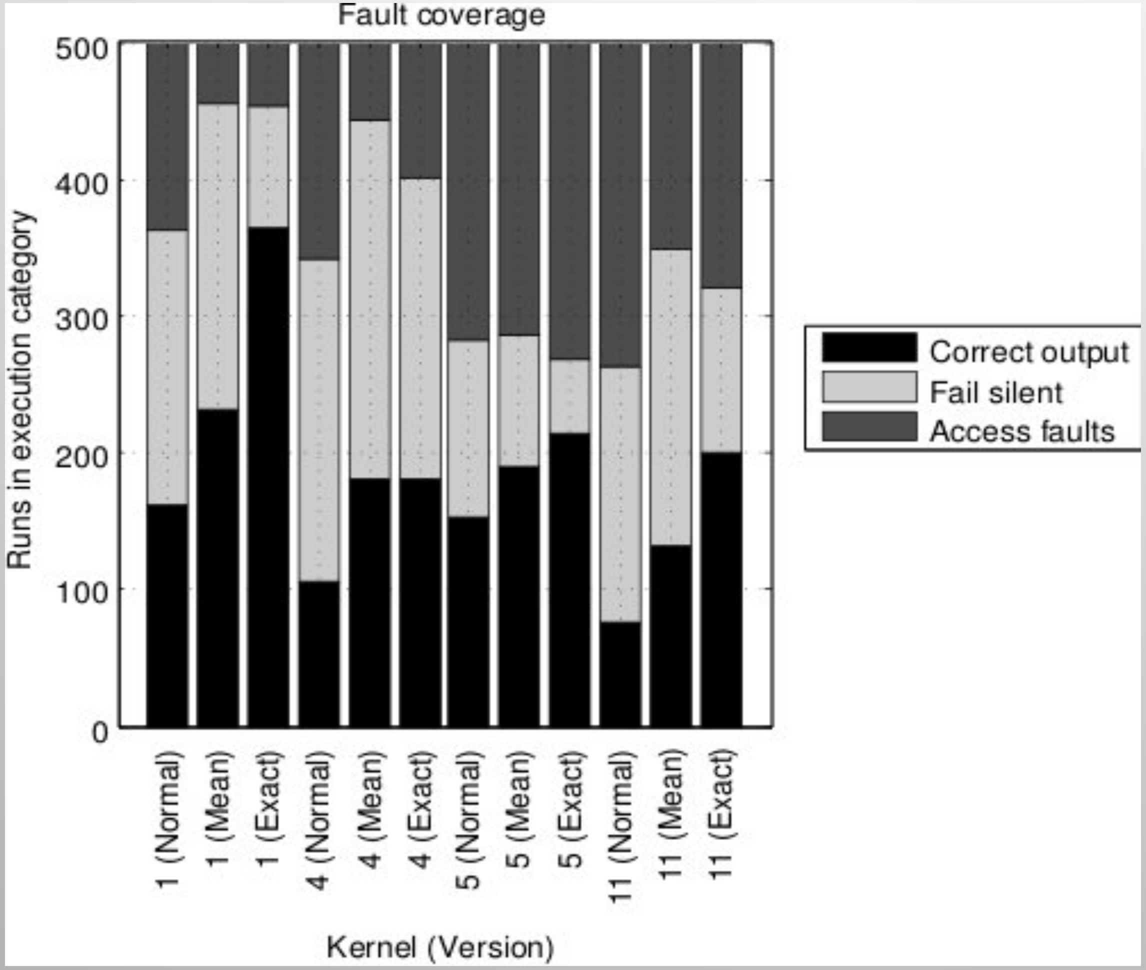
	1-D 1-Point	1-D 3-Point	2-D 5-Point
Iteration stride = 1			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	17.33%	29.91%	30.19%
double	27.55%	22.27%	22.60%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	13.87%	25.58%	25.03%
double	17.43%	19.97%	17.37%
Iteration stride = 8			
Array size: 1 million for 1-D, 4096x4096 for 2-D			
float	8.36%	19.12%	25.39%
double	5.10%	6.33%	5.44%
Array size: 16 million for 1-D, 16Kx16K for 2-D			
float	3.57%	10.30%	14.54%
double	0.05%	0.80%	1.59%

- Livermore kernel
  - Kernel 1 (Hydro fragment) – 20%
  - Kernel 4 (Banded linear equations) – 40%
  - Kernel 5 (Tri-diagonal elimination) – 26%
  - Kernel 11 (First sum) – 2%

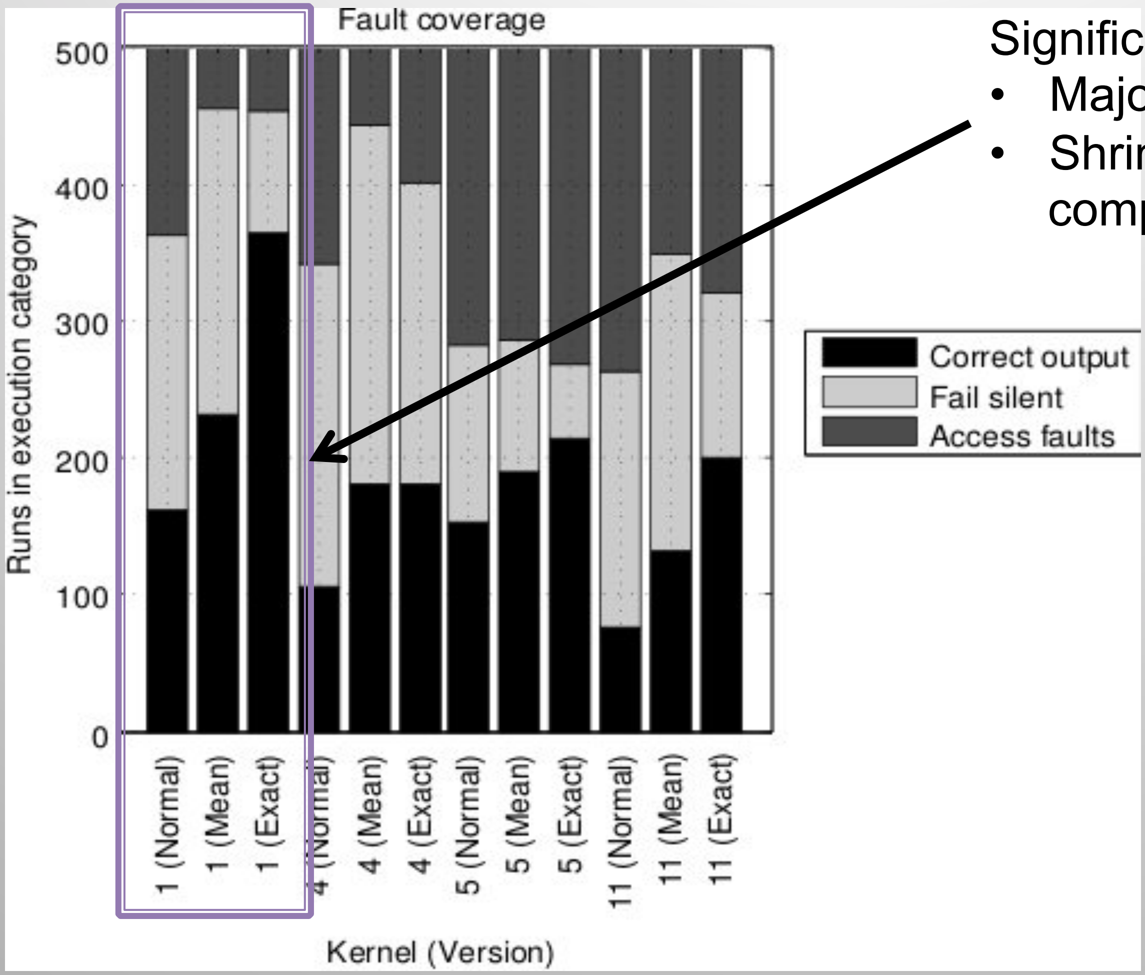
Results: fault coverage and effectiveness

- Benchmarks: Livermore Loops suite
  - kernels #1, #4, #5, and #11
  - Three versions for each kernel:
    - Original unprotected code
    - Mean: TMR (using baseline DMR) and mean voting is added.
    - Exact: TMR (using baseline DMR) and exact majority voting is added.
- Fault injection: Intel Pin tool
  - Training runs: record correct instruction counts and output
  - Fault injection runs (500 times): flip a random bit of input general purpose/floating point register of a random instruction
- Exit condition categories the execution
  - Correct result, Access fault (invalid memory access), Fail silent, Invalid instruction, Invalid arithmetic operation

Results (cont.) - Fault coverage



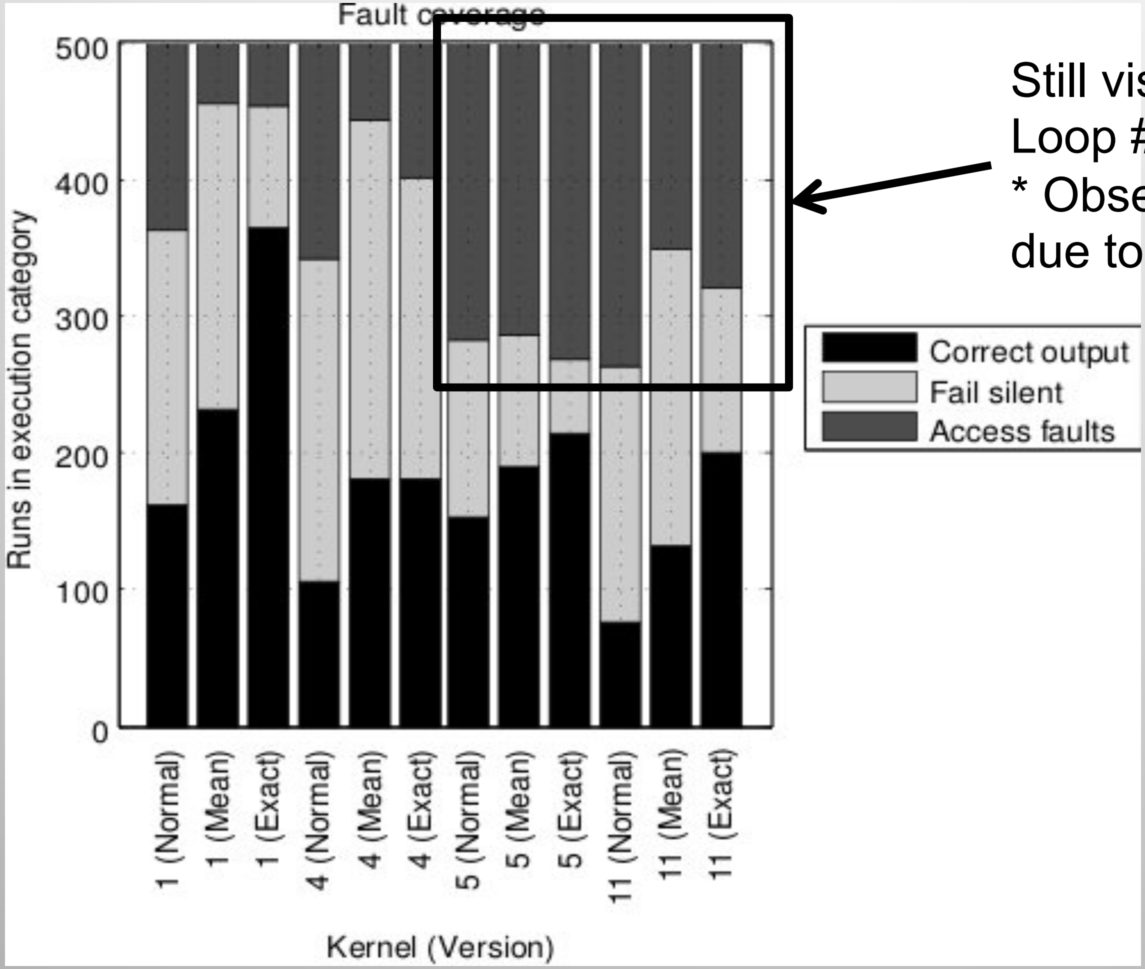
Results (cont.) - Fault coverage



Significant improvements: Loop#1

- Majority voting > Mean voting
- Shrinking fail silent (soft errors) compared to unprotected version

Results (cont.) - Fault coverage



Still visible improvements for Loop #5 and #11
* Observed many access faults due to many array index corruptions

Summary

- Fault handling via source-level code transformation
  - + Low cost and flexible
  - + Keeps programmer in-the-loop
  - - Can't specify low-level details
- Feasibility: work with compiler optimizations.
  - CSE issues can be *overcome* with careful program transformation
- Overhead: N redundant executions != Nx slower
  - N-2 redundancy on demand
  - Hide overhead within latencies of original code (could be plenty for Exascale!)
- Effectiveness:
  - In both fault detection and handling

Future work

- Using multithreading for duplicated work:
  - thread vs. instruction/statement level redundancy
- Include more fault handling policies
- More ways to live with compiler optimizations (CSE)
  - transformation at binary level
- Automatically identify critical code portions for added resilience
  - Probabilistic model of operations, sensitivity to input characteristics

Thank You!

- Questions?